МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт

"Розробка програм для симетричних багатопроцесорних обчислювальних систем з використанням OPENMP у середовищі VISUAL STUDIO"

з дисципліни "Архітектура обчислювальних систем"

для студентів спеціальностей
122 – Комп’ютерні науки та інформаційні технології,
124 – Системний аналіз,186 – Видавництво та поліграфія

Затверджено
редакційно-видавничою
радою університету,
протокол №2 від 23.06.17

Харків
НТУ «ХПІ»
2018

Методичні вказівки до лабораторних робіт «Розробка програм для симетричних багатопроцесорних обчислювальних систем з використанням OPENMP у середовищі VISUAL STUDIO» з дисципліни "Архітектура обчислювальних систем" для студентів спеціальностей 122 – Комп’ютерні науки та інформаційні технології, 124 – Системний аналіз,186 – Видавництво та поліграфія. / уклад. Ю. М. Кожин, О. М. Малих, В. П. Прокопенков. – Харків.: НТУ “ХПІ”, 2018. – 52с.

Укладачі: Ю.М. Кожин,
О.М. Малих,
В.П. Прокопенков

Рецензент М.І. Безменов

Кафедра системного аналізу і інформаційно–аналітичних технологій

Вступ

Одним із способів розробки програм для комп'ютерів із загальною пам'яттю є технологія *OpenMP*. Інтерфейс *OpenMP* розроблений як стандарт для програмування на масштабованих симетричних багатопроцесорних системах *SMP* (*SSMP, ccNUMA* та інших).

У підсистему програмування *OpenMP* входять набір директив компілятора, допоміжні функції і змінні середовища.

OpenMP реалізує паралельні обчислення за допомогою богатопоточності, в якій "головний" потік створює набір "підпорядкованих" потоків, і завдання розподіляються між ними. Передбачається, що потоки виконуються паралельно на машині з декількома процесорами.

Технологія розробки додатків з використанням *OpenMP* полягає в тому, що за основу береться послідовна програма, в яку для створення паралельної версії користувач вставляє директиви формування додаткових потоків. При цьому користувач має один варіант програми як для паралельного, так і для послідовного виконання.

Дані методичні вказівки укладені з метою допомогти студентам оволодіти технологією застосування підсистеми *OpenMP* при розроблені програм з паралельним виконанням у середовище програмування *Visual Studio.*

1. ОРГАНІЗАЦІЯ ПАРАЛЕЛЬНОГО ВИКОНАННЯ ПРОГРАМ З ВИКОРИСТАННЯМ ПІДСИСТЕМИ *OPENMP*

Мета роботи: отримання навичок розробки програмного забезпечення з використанням *OpenMP*.

1.1. Розробка програм у симетричних багатопроцесорних системах

1.1.1. Склад підсистеми OpenMP

OpenMP використовується для програмування в *SMP*-системах (симетричні багатопроцесорні системи) в моделі загальної пам'яті (*shared memory model*).

OpenMP реалізує паралельні обчислення за допомогою багатопоточності: "головний" потік створює набір "підпорядкованих" потоків, і виконання програми розподіляється між окремими потоками. Передбачається, що потоки виконуються паралельно на машині з декількома процесорами.

До складу підсистеми *OpenMP* входять:

- директиви компілятора (препроцесора);
- допоміжні функції;
- змінні середовища (системні змінні).

Компілятор інтерпретує директиви *OpenMP* і створює паралельний код. При використанні компіляторів, які н е підтримують *OpenMP*, директиви *OpenMP* ігноруються без додаткових повідомлень.

Компілятор з підтримкою *OpenMP* має макрос *_OPENMP*, що визначає число у форматі *yyyymm*, де *yyyy* та *mm* – цифри року і місяця, коли був прийнятий підтримуваний стандарт *OpenMP*.

Для організації паралельної роботі разом з директивами компілятора можуть бути використані функції налаштування середовища виконання.

Функції бібліотеки *OpenMP* описані в заголовному файлі *omp.h*

1.1.2. Настройка компілятора

Для використання механізмів *OpenMP* необхідно настроїти компілятор. Налаштування здійснюється через пункт меню "Проект" –"Властивості"–"C/C++"–"Мова" установленням прапора "Підтримка *OpenMP*" у відповідне значення (рис.1.1).

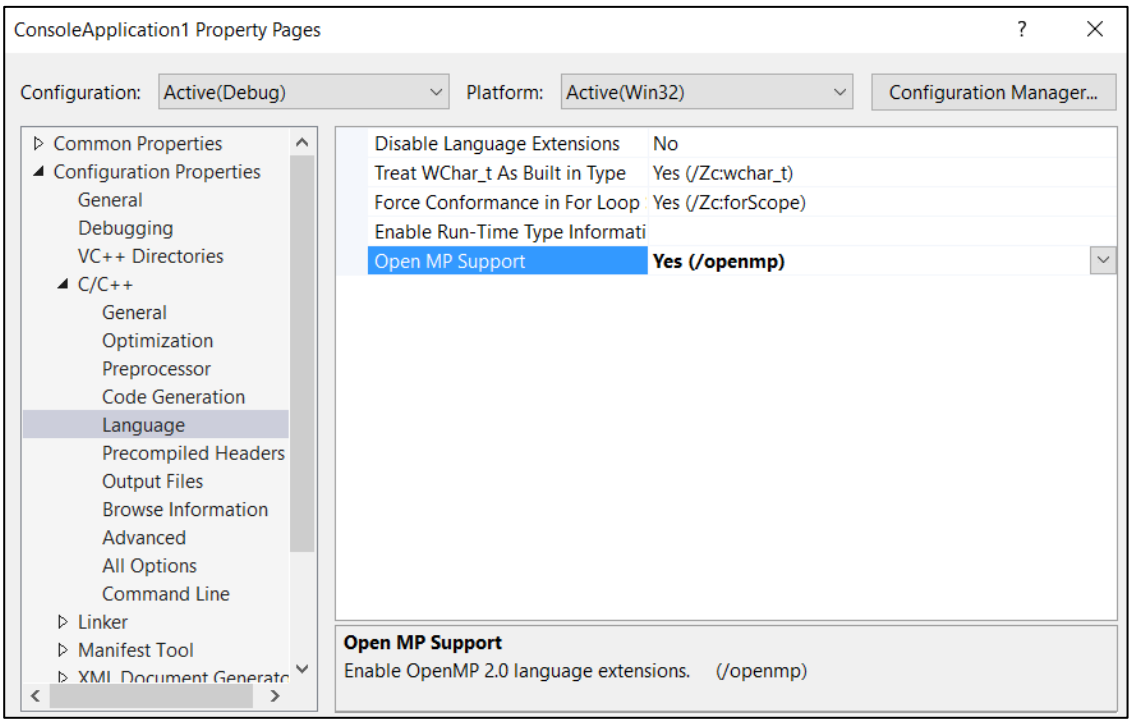

Рисунок 1.1 – Установлення режиму використання *OpenMP*

1.1.3. Модель паралельної програми

Створення паралельної програми в *OpenMP* виконується явно за допомогою вставки в текст спеціальних директив і виклику допоміжних функцій. При використанні *OpenMP* передбачається *SPMD*-модель (*Single Program Multiple Data*) паралельного програмування, у рамках якої для усіх паралельних потоків використовується той самий же код програми (рис.1.2).

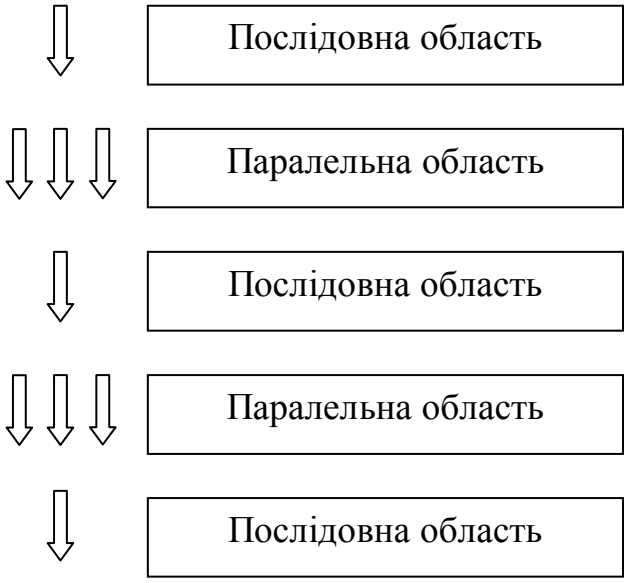

Рисунок 1.2 – Виконання програми з використанням *OpenMP*

Програма розпочинається з послідовної області. При вході в паралельну область породжується ще деяке число потоків виконання програми, між якими надалі розподіляються частини програми. Після закінчення паралельної області усі потоки, окрім одного, завершуються, і починається послідовна область. У програмі може бути будь-яка кількість паралельних і послідовних областей. Паралельні області можуть бути також вкладеними одна в одну.

На відміну від процесів, породження потоку є відносно швидкою операцією, тому часті породження і завершення потоків не так сильно впливають на час виконання програми. Проте при великій кількості потоків ефективність багатопотокового виконання програми може бути значно знижена через накладні витрати на організацію перемикання процесора обчислювача між потоками.

1.1.4. Директиви OpenMP

Значна частина функціональності *OpenMP* реалізується за допомогою директив компілятора. Вони мають бути явно вставлені користувачем, що

дозволить виконувати програму в паралельному режимі. Директиви *OpenMP* мають синтаксис:

#pragma omp *директива* [*параметр*[, *параметр*...]

{

Блок операторів мови С (С++)

} ,

де *директива* – директива *OpenMP*, а *параметр* – параметри директиви.

Дія директиви поширюється виключно на блок операторів у фігурних дужках, який повинен розміщуватися відразу за директивою.

Порядок параметрів в описі директиви не має значення, в одній директиві деякі параметри можуть зустрічатися кілька разів. Параметри можуть мати список змінних.

Усі директиви *OpenMP* можна розділити на 3 категорії:

- визначення паралельної області;
- розподіл роботи;
- синхронізація роботи потоків.

Для організації паралельної роботи разом з директивами компілятора можуть бути використані функції налаштування середовища виконання.

Функції бібліотеки *OpenMP* описані в заголовному файлі *omp.h*.

1.1.4.1. Директива початку паралельної області

Паралельна область задається за допомогою директиви *parallel*.

#pragma omp parallel

{

…

}

При вході в паралельну область породжуються нові потоки, кожен з них отримує свій унікальний номер. Потік, що породжує, отримує номер 0 і стає основним потоком групи ("майстром"). Інші потоки отримують як номер цілі числа від 1 до встановленого значення (установлення кількості потоків

здійснюється за допомогою функції *omp _ set _ num _ threads*). Кількість потоків, що виконують цю паралельну область, залишається незмінною до моменту виходу з області. При виході з паралельної області здійснюється неявна синхронізація і знищуються усі потоки, окрім основного (рис.1.3).

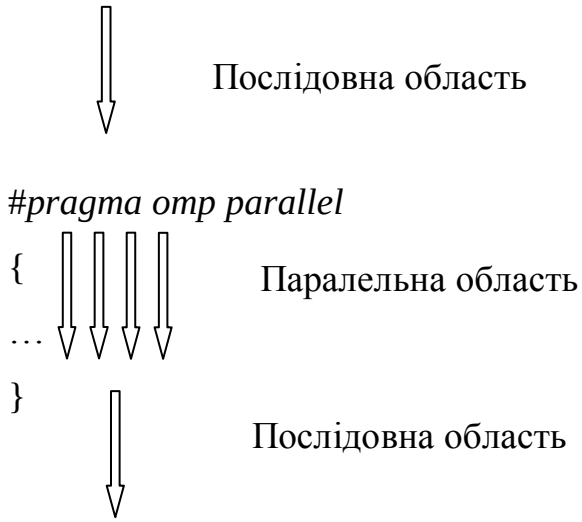

Рисунок 1.3 – Створення потоків за допомогою директиви *parallel*

Усі породжені потоки виконують той же самий код, що відповідає паралельній області. Паралельні області можуть бути вкладеними. За замовчуванням вкладена паралельна область виконується тільки одним потоком.

Функція *omp _ set _ nested (int nested*) дозволяє (*nested*=1) або забороняє (*nested* =0) вкладений паралелізм. Якщо вкладений паралелізм дозволений, то кожен потік, в якому зустрінеться опис паралельної області, породить для її виконання нову групу потоків. Потік, що породжує інші, стане в новій групі потоком-майстром.

1.1.4.2. Директива *single*

Якщо в паралельній області яка-небудь ділянка коду має бути виконана лише один раз (одним потоком), то його треба виділити директивою *single*.

#pragma omp single

{

…

}

Який саме потік виконуватиме виділену ділянку програми – невідомо. Один із запущених раніше потоків виконуватиме цей фрагмент програми, а усі інші потоки чекатимуть його завершення, якщо тільки не вказаний параметр *nowait* директиви (рис.1.4).

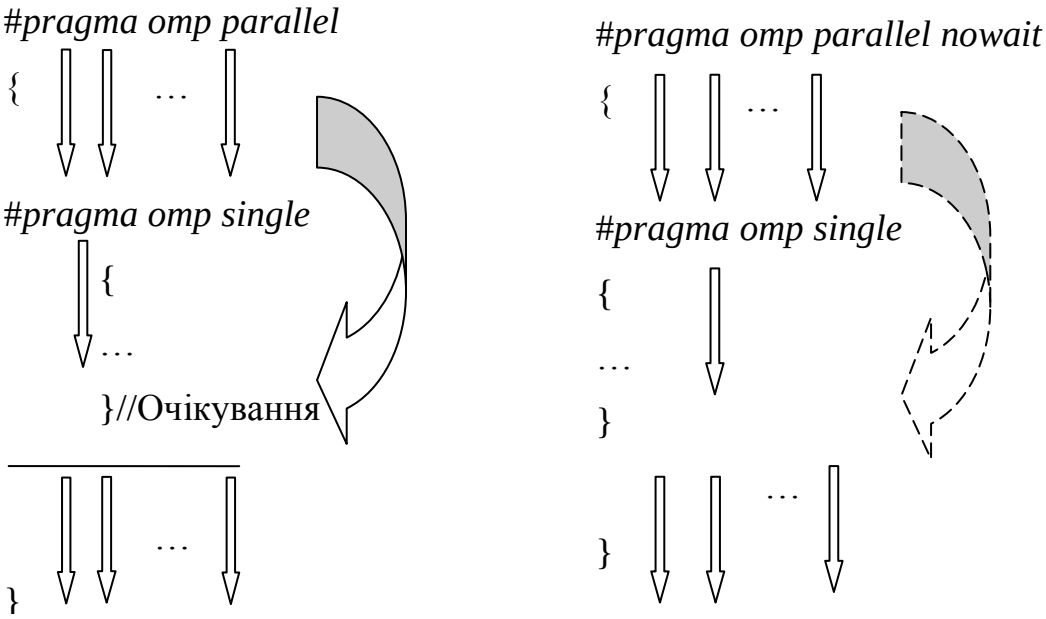

Рисунок 1.4 – Організація ділянки програми для виконання одним потоком в паралельній області

Необхідність використання директиви *single* часто виникає при роботі із загальними для потоків змінними або для видачі проміжних даних.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
#pragma omp parallel
  {
#pragma omp single nowait
    { … }
```

cout <<” Видача проміжних даних”;… }

}

1.1.4.3. Директива *master*

Директива *master* виділяє ділянку коду, який буде виконаний тільки основним потоком (рис. 1.5). Інші потоки пропускають цю ділянку і продовжують роботу з оператора, розташованого слідом за ним. Неявної синхронізації ця директива не здійснює.

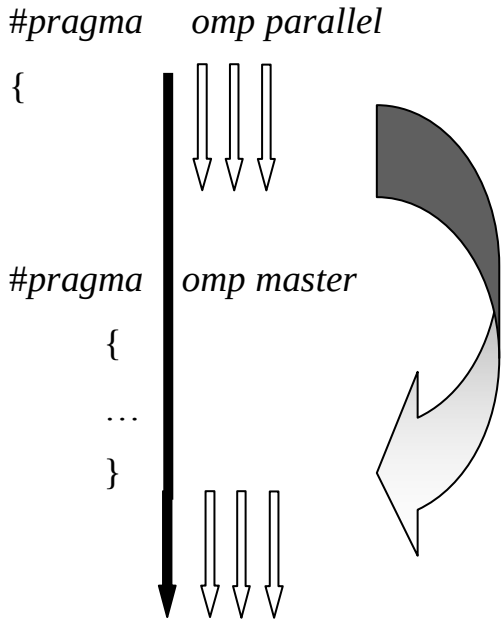

Рисунок 1.5 – Виділення ділянки програми для виконання головним потоком

Нижче наведений приклад програми з створення паралельної ділянки з чотирма потоками. Змінна *n* є локальною для кожного потоку (у кожному потоці є копія змінної). Для головного потоку значення змінної дорівнює 2, для інших потоків 1.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
 int n;
```

```
#pragma omp parallel num_threads(4) private(n)
 {
 n=1;
#pragma omp master
 {
 n=2;
 }
…
printf("Значення n: %d\n", n);
 }
}
```

1.1.5. Параметри паралельної області

Директива оголошення паралельної області допускає використання різних параметрів. Усі параметри паралельної області можна розділити на три групи:

- управління входом у паралельну область (*if*);
- завдання числа потоків, що виконуються в паралельній області (*num_threads*);
- використання зовнішніх і локальних змінних у рамках паралельної області (*private*, *firstprivate*, *copyin*, *reduction*, *shared*, *default*)

1.1.5.1. Визначення умови входу в паралельну область

Для управління входу в паралельну область використовується директива *parallel* з параметром *if*

#pragma omp parallel if (*умова*) .

Умова в директиві може містити будь-який вираз з використанням констант і змінних програми (глобальних та локальних). Якщо вираз дає істинне значення, здійснюється вхід у паралельну область та формування заданого числа потоків, що виконуються в паралельній області. В іншому

випадку директива не формує паралельну область, і обробка програми продовжується, як і раніше (один потік виконує дану ділянку програми).

У наведеному нижче прикладі визначається номер версії підсистеми *OPENMP*. Якщо номер версії підсистеми має рік розробки раніше 2002 року, ділянка програми виконується як послідовна.

```
#include <stdio.h>
#include <omp.h>
int main( )
{
        int v = _OPENMP;
        printf_s("version %d ",v);
#pragma omp parallel if( v/100>=2002 )
        {
…
        }
return 0;
}
```

1.1.5.2. Встановлення кількості потоків паралельної області

При оголошенні паралельної області можна явно вказати кількість потоків, які будуть виконувати паралельну область. Для цього використовують параметр *num_threads*

#pragma omp parallel num_threads (кількість потоків) .

Аргумент параметра, що задає кількість потоків, повинен мати позитивне значення. Рекомендується кількість потоків задавати не більше, ніж число процесорних елементів.

Якщо параметр *num_threads* не задано, вибирається останнє значення, встановлене за допомогою функції *omp_set_num_threads*( ) або значення за замовчунням .

1.1.6. Організація доступу до змінних

У паралельнії області може здійснюватися доступ до змінних, описаних у зовнішній області. У такому разі в паралельній області може бути призначений режим доступу. Режим доступу визначає створення копій змінних для окремих потоків і правило присвоювання початкового значення.

В паралельній області можуть бути оголошені власні змінні. Такі змінні не розділяються між потоками, і зміни цих змінних в одному потоці не впливають на значення відповідних змінних іншого потоку.

1.1.6.1. Локальні копії змінних потоку.

Для створення локальних копій зовнішніх змінних для кожного потоку використовуються параметри *private, firstprivate, copyin.*

Кожен потік може змінювати значення локальних копій змінних. Ці зміни не впливають на значення змінних в інших потоках. Після закінчення потоку значення локальних копій не зберігається.

Параметр *private* (список) – задає список змінних, для яких породжується локальна копія в кожному потоці

#pragma omp parallel private (список) .

Початкове значення локальних копій змінних зі списку не визначено.

Параметр *firstprivate* задає список змінних, для яких створюються копії в кожному потоці

#pragma omp parallel firstprivate (*список*) .

Локальним копіям змінних присвоюються значення, задані в попередньому потоці.

Параметр *copyin* – задає список змінних, оголошених як *threadprivate*, які при вході в паралельну область мають значення відповідних змінних попереднього потоку.

#pragma omp parallel copyin (список)

На відміну від параметра *firstprivate* параметр *copyin* дозволяє створювати копії статичних і глобальних змінних.

Перед використанням *copyin* змінна повинна бути оголошена як *threadprivate* за допомогою директиви *pragma omp threadprivate*. Наприклад:

static int k = 0;

#pragma omp threadprivate (*k*)

*#pragma omp parallel num_th*reads (3) *copyin*(*k*)

{

…// для кожного потоку локальна копія змінної *k*

}

1.1.6.2. Використання зовнішніх змінних

Для доступу до зовнішніх змінних використовується параметр доступу *shared*

#pragma omp parallel shared (список) .

Параметр *shared* задає список змінних, загальних для всіх потоків. Змінні, оголошені з доступом *shared*, слід використовувати для зчитування даних. Збереження нового значення для змінних типу *shared* може призводити до непередбачуваних наслідків, оскільки невідомо, в якому порядку будуть виконуватися потоки і, відповідно, яке значення буде записане в поточний момент часу.

1.1.7. Формування вихідного значення локальних змінних потоку

Значення деяких змінних різних потоків можуть бути об’єднані в одне значення після закінчення потоків. Для створення списку таких змінних і завдання операції з обчислення загального значення використовується параметр *reduction* паралельної області

#pragma omp parallel reduction (*операція* : *список*)

Операція може являти собою одне з можливих значень: + (додавання), * (множення), & (побітова операція «і»), | (побітова операція «або»),

^ (побітова операція «сума за модулем 2»), && (логічне «і»), || (логічне «або»).

Для кожної змінної зі списку створюється локальна копія в кожному потоці. Після виконання паралельної області над локальними копіями змінних і значенням зовнішньої змінної виконується задана операція.

1.1.8. Звернення до локальних і загальних змінних з паралельної ділянки

Наведений нижче приклад використовує локальні і загальні змінні в потоках *OMP*.

```
#include <omp.h>
int y=1;
#pragma omp threadprivate ( y )
int main()
{
        int i;
        int j;
        int m;
        static int k = 0;
#pragma omp threadprivate ( k )
        m =0;
        i = –1;
        j = –5;
printf_s("main begin j= %d i= %d k= %d y = %d\n", j, i, k, y);
#pragma omp parallel num_threads ( 3 ) private ( j ) firstprivate ( i ) \
                                        copyin ( k , y ) reduction ( + : m )
{
int n;
n= omp_get_thread_num();
```

```
j = n;
printf_s("\n potok %d begin j= %d i= %d   k= %d y = %d\n", n,j, i, k,y );
i = n; k = n; y ++; m = n;
printf_s("potok %d m=%d j= %d i= %d k= %d y = %d\n\n", n, m, j, i, k, y);
}
printf_s("\n m=%d \n", m);
printf_s("main end j= %d i= %d   k= %d y = %d\n", j, i, k, y);
_wsystem(L"pause");
 return 0;
}
```

Змінна *j* використовується як локальна змінна. Вона має власну копію для кожного потоку, значення змінної *j* не зберігається після закінчення паралельної ділянки.

Змінна *i* використовується як локальна змінна. Вона має власну копію для кожного потоку. Початкове значення змінної визначається перед початком паралельної ділянки. Зміна значення змінної *i* не зберігається після закінчення паралельної ділянки.

Змінна *k* – статична змінна функції *main*, а змінна *y* – глобальна змінна. За допомогою директиви *pragma omp threadprivate* для цих змінних установлюється можливість створення копій в паралельній ділянці. Зміна значень змінних *k* та *y* не зберігається після закінчення паралельної ділянки.

Змінна *m* має копію для кожного потоку. Значення копій змінної підсумовується перед закінченням паралельної ділянки.

1.2. Допоміжні функції

1.2.1. Визначення часу виконання ділянки програми

В *OpenMP* передбачені функції для роботи з системним таймером.

Функція *omp_get_wtime*(*void*) повертає астрономічний час у секундах (дійсне число подвійної точності), що минув з деякого моменту в минулому.

Функція *double omp_get_wtick*(*void*) повертає точність таймера в секундах.

Використання функцій виміру часу наведено нижче.

```
#include <stdio.h>
#include <omp.h>
int main( int argc, char *argv[] )
{  double begin_time, end_time, tick;
  begin_time = omp_get_wtime ( );
 …      // виконання потоку
end_time = omp_get_wtime ( );
tick = omp_get_wtick ( );
printf("Час %lf\n", end_time - begin_time);
printf("Точність таймера %lf\n", tick);
}
```

1.2.2. Функції бібліотеки OpenMP

Бібліотека функцій *OMP* містить ряд додаткових функцій (табл. 1.1).

Таблиця 1.1 – Додаткові функції *OMP*

Функція	Опис
omp_set_num_threads(*int N*)	Установлення числа потоків (*N*)
int omp_get_num_threads ( )	Отримання числа потоків
int mp_get_max_threads ( )	Отримання максимального числа потоків з урахуванням можливостей апаратури
int omp_get_thread_num ( )	Номер поточного потоку
int omp_get_num_procs ( )	Число фізичних процесорів в обчислювачі

Продовження табл. 1.1

omp_set_dynamic (*int D*)	Установлення прапора динамічного вибору

	числа потоків при виконанні паралельних ділянок
Функція	Опис
int omp_get_dynamic( )	Перевірка прапора динамічного вибору числа потоків
omp_set_nested (*int N*)	Дозвіл (N = 1) / заборона (N = 0) прапора формування вкладеного потоку
int omp_get_nested ( )	Значення прапора дозволу вкладеного потоку
int omp_in_parallel ( )	Перевірка виконання програми у паралельному потоці

1.3. Порядок виконання роботи

1. Розробити програму для обчислення значень функції на заданому інтервалі з послідовним варіантом виконання відповідно до завдання.
2. Виділити ділянки з можливим паралельним виконанням.
3. Вставити в послідовний варіант директиви для паралельного виконання програми.
4. Вставити в текст програми функцію виміру часу виконання.
5. Здійснити трансляцію програми в послідовному варіанті.
6. Зняти час виконання послідовної програми залежно від розмірності задачі.
7. Налаштувати транслятор системи програмування на використання режиму OpenMP.
8. Здійснити трансляцію програми для роботи в паралельному режимі.
9. Зняти час виконання програми в паралельному режимі залежно від розмірності задачі.
10. Перевірити роботу програми залежно від кількості потоків у паралельних областях.

11. Побудувати графіки для часу виконання завдання в послідовному і паралельному режимах. Результати пояснити.

Таблиця 1.2 – Варіанти завдань

Варіант	Формула для обчислення	Інтервали
1	$y = \begin{cases} 1 - \sqrt{\lvert \cos(2x) \rvert} & x > a \\ x^2 - x & b < x < a \\ 1 + x^2 & x < b \end{cases}$	x=0÷3 a=2,5 b=1
2	$y = \begin{cases} 2x & x > a \\ 1 - \ln\lvert 1 + x^2 \rvert & b < x < a \\ e^{-x} & x < b \end{cases}$	x= – 0,5 ÷5 a=4,5 b=0
3	$y = \begin{cases} \sqrt{\ln(x^2 - 1)} & x > a \\ -2x^3 & b < x < a \\ e^{\sin(x)} & x < b \end{cases}$	x= – π/2÷π a=2 b=0
4	$y = \begin{cases} 1 + x & x > a \\ 1 - x^5 & b < x < a \\ x \ln\lvert \sin(x) \rvert & x < b \end{cases}$	x= – π÷π/2 a=2,5 b=0
5	$y = \begin{cases} x - 2 & x > a \\ 1 + x^2 & < x < a \\ x \ln\lvert \cos(x) \rvert & x < b \end{cases}$	x= – π/2÷2π a=2,5 b=0
6	$y = \begin{cases} 1 + 3x & x > a \\ e^{-2x} & b < x < a \\ \cos(2x) & x < b \end{cases}$	x=– π/2÷2π a=4,5 b=0
7	$y = \begin{cases} \sqrt{\mathrm{tg}(x^2 - 1)} & x > a \\ -2x & b < x < a \\ e^{\cos(x)} & x < b \end{cases}$	x= – 2,5÷5 a=4 b=0

Продовження табл. 1.2

Варіант	Формула для обчислення	Інтервали

8	$y = \begin{cases} 1 + \sqrt{\lvert \sin(2x-1) \rvert} & x > a \\ x^2 - x & b < x < a \\ 1 + x^2 & x < b \end{cases}$	x= – π/2÷2π a=2,5 b=0
9	$y = \begin{cases} 1 - \sqrt{\lvert \cos(2x) \rvert} & x > a \\ 1 + 3\sin(x^2) & b < x < a \\ 1 + x^2 & x < b \end{cases}$	x= – π÷π/2 a=2,5 b=0
10	$y = \begin{cases} 1 - \sqrt{\lvert \mathrm{tg}(2x-3) \rvert} & x > a \\ x^2 - \sin(x) & b < x < a \\ 1 - x^2 & x < b \end{cases}$	x= – 0,5÷5 a=4,5 b=0

Контрольні питання

1. Для яких систем програмування використовується підсистема *OpenMP*?

2. За допомогою чого в *OpenMP* реалізується паралельне виконання програми?

3. Що входить до складу *OpenMP*?

4. Який режим компілятора потрібно встановити для використання підсистеми *OpenMP*?

5. Чи завжди спостерігається підвищення ефективності виконання програми при використанні *OpenMP*?

6. На які категорії можна розділити директиви *OpenMP*?

7. За допомогою якої директиви задається паралельна область виконання програми?

8. Яка кількість потоків формується при вході в паралельну область?

9. Які номери отримують потоки при вході в паралельну область?

10. Чи може паралельна область містити оголошення вкладеної паралельної області?

11. Яка кількість потоків формується при вході у вкладену паралельну область?

12. Для чого використовується директива *single*?

13. Як задається умова для входу в паралельну область?

14. Як організується доступ до загальних і локальних змінних у паралельній області?

15. Для чого використовується параметр *reduction* в директиві *parallel*?

16. Які операції допустимі в параметрі *reduction*?

17. За допомогою якої функції можна здійснити вимір часу виконання ділянки програми?

15. Для чого використовується параметр *reduction* в директиві *parallel*?

16. Які операції допустимі в параметрі *reduction*?

2. ПАРАЛЕЛЬНЕ ВИКОНАННЯ ПРОГРАМ З ВИКОРИСТАННЯМ *OPENMP*

Мета роботи: освоєння прийомів виділення ділянок програми для організації потоків з паралельним виконанням.

2.1. Паралельне виконання програм

2.1.1. Організація паралелізму на основі номера потоку

Всі потоки в паралельній області нумеруються послідовними цілими числами від 0 до $N - 1$, де N – кількість потоків, які виконуються в даній області.

Інформацію про номер потоку і загальне число запущених потоків можна використовувати для розподілу роботи між потоками.

Загальну кількість потоків, що запускається в паралельній області, можна отримати за допомогою функції *int omp_get_num_threads*( ).

Виклик функції *int omp_get_thread_num* () дозволяє в потоці отримати свій унікальний номер в поточній паралельної області.

Наприклад, необхідно здійснити пошук максимального значення елемента в масиві чисел.

Увесь масив можна розбити на кілька частин і в кожній частині виконати пошук локального максимального елемента, після чого вибрати максимальне значення з отриманих локальних максимумів.

Нижче наведено текст програми для пошуку максимального елемента в масиві *Mass* розмірністю 100 × 100 елементів. Масив можна розбити на чотири частини по рядках (по 25 рядків у кожній частині). Кожен потік за номером визначає межі пошуку локального максимуму. Локальні максимуми записуються у допоміжний масив *maximum*.

```
Int Mass[M ][M];
…
        int th; // число потоків
        th = 4;
int GlobMax=0;
        int *maximum = new int[th];
#pragma omp parallel num_threads(th)
        {
                int n,k;
                int beg;
                th = omp_get_num_threads ( );
                n = omp_get_thread_num ( );
                int maxN;
                beg = M / th*n; //  номер строки початку блока
                maxN = a [ beg ][0];
                for (k = beg; k < M / th*( n + 1 ); k++)
```

```
                    for (int j = 0; j < M; j++)
                        if (Mass[k][j]>maxN)
                            maxN = Mass[k][j];
                maximum[n] = maxN;
        }
    GlobMax      = max (maximum[0], maximum[1]);
    int GlobMax1 = max (maximum[3], maximum[2]);
    GlobMax=max (GlobMax, GlobMax1);
```

2.1.2. Паралельні секції

Другим способом організації паралелізму є використання директиви *sections*.

Директива *sections* задає ділянку в послідовній програмі, в якій можна виділити кінцеве число незалежних один від одного секцій коду. Директива *section* задає окрему секцію для потоку.

```
#pragma omp parallel sections [параметр[, параметр]...]
{  #pragma omp section
   {
// перша секція
    }
   #pragma omp section
   {
// друга секція
   }
…
}
```

Які саме потоки будуть задіяні для виконання будь-якої секції, не визначено. Якщо кількість потоків більше кількості секцій, то частина потоків для виконання даного блока секцій не буде задіяна. Якщо кількість

потоків менше кількості секцій, то деякі потоки будуть виконувати код декількох секцій.

Директива *section* може мати параметри аналогічно директиві *parallel*, а також додаткові:

lastprivate (список) – змінним, перерахованим у списку, присвоюється результат, отриманий в останній секції;

nowait – дозволяє потокам, які дійшли до кінця своїх секцій, продовжити виконання без синхронізації з іншими.

Директива *section* задає ділянку коду всередині секції *sections* для виконання одним потоком.

#pragma omp section

Перед першою ділянкою коду у блоці *sections* директива *section* не обов'язкова.

Використання директиви *sections* можна розглянути на такому прикладі.

Для заданого значення λ необхідно перевірити, чи є вектор $X = \{x_i | i = 1, n\}$ власним вектором для матриці $A = \{a_{i,j} | i, j = 1, n\}$. Для цієї мети перевіряється істинність виразу

$$(A - \lambda E)X = \lambda X ,$$

де λ – власне значення матриці A, E – одинична матриця.

Нехай матриця A зчитується з файлу, а вектор X вводиться з клавіатури.

Розмірність матриці n, власне число λ і точність обчислення передаються у програму через параметри функції main.

Тоді зчитування матриці A і обчислення виразу $(A - \lambda E)$ можна здійснити в одному потоці, а зчитування вектора X і множення його на власне число – в іншому потоці.

#include "*stdafx.h*"

```
#include <stdlib.h>
#include <omp.h>
#include <fstream>
#include <iostream>
using namespace std;
int main( int argc, char *argv [ ] )
        {
                float ** a;
                float * x;
                float * Lx;
                float eps = atof ( argv [ 3 ]);
                int flag = 0;
                int M = atoi ( argv [ 2 ]);
                int L  = atof ( argv [ 1 ]);
                ifstream file ("input.txt");
                // кількість потоків – 2
#pragma omp  parallel sections num_threads (2)
                { // Перша секція
#pragma omp section
                { x = new float [ M ];
                for (int i = 0; i < M; ш++)
                {
                        cin >> x [ i ];
                        Lx [ i ] = x [ i ] * L;
                }
                }
#pragma omp section
                { // Друга секція
                        a = new float * [ M ];
```

```
                for (int i = 0; i < M; i++)
                {
                        a[i] = new float [ M ] ;
                        for (int j = 0; j < M; j++)
                        {
                                file >> a [ i ] [ j ];
                        }
                        a [ i ] [ i ] = a [ i ] [ i ] – L;
                }
        }
    }// закінчення паралельної частини
// Послідовна ділянка
    for (int i = 0; i < M; i++)
    {
        float z = 0;
        for (int j= 0; j < M; j++)
        {
            z = z + a [ i ] [ j ]* x [ j ];
        }
        if (abs(z – Lx[i]) > eps)
        {
            flag = 1;
            break;
        }
    }
    if (0  = =  flag)
        cout << "Вектор X не є власнім " << endl;
    else
        cout << "Вектор X – власний " << endl;
```

return 0;

}

2.1.3. Паралельні цикли

Якщо в паралельній області зустрівся оператор циклу, то, відповідно до загального правила, він буде виконаний всіма потоками поточної групи, тобто кожен потік виконає всі ітерації даного циклу. Для розподілу ітерацій циклу між різними потоками можна використовувати директиву *OMP for*.

На вид паралельних циклів накладаються жорсткі обмеження:

- виконання програми не повинно залежати від того, який саме потік буде виконувати будь-яку ітерацію циклу;
- не можна використовувати побічний вихід з циклу (вихід за допомогою оператора *break* або *goto*);
- кількість ітерацій має бути відома перед початком циклу (початкове значення, кінцеве значення змінної циклу і величина зміни лічильника циклу не повинні змінюватися всередині циклу);
- цикл for повинен мати вигляд

for (i = N; i {<,>,=,<=,>=} M; i {+,-}= K) ,

де *i* – параметр циклу (змінна цілого типу); *N, M* і *K* –- цілочисельні вирази; фігурні дужки визначають можливі варіанти операцій порівняння і зміни параметра циклу.

Ітеративна змінна циклу повинна бути локальною. Якщо вона визначена як загальна, то змінна циклу неявно робиться локальною при вході в цикл. Після завершення циклу значення змінної циклу не визначено, якщо вона не вказана у списку *lastprivate*.

Директива організації паралельної роботи циклу має вигляд:

#*pragma omp for* [*параметр* [, *параметр*]...]

Ця директива належить до циклу, що йде безпосередньо слідом за даною директивою.

Директива *OMP for* може мати режими *private, firstprivate, reduction lastprivate* аналогічно директиві *sectons* для завдання режимів доступу до змінних, а також додаткові параметри виконання циклу:

schedule (тип [, *L*]) – спосіб розподілу *L* ітерацій між потоками;

ordered – потоки будуть запускатися в порядку послідовного циклу.

За замовчуванням кожен потік запускається в довільному порядку. При цьому кожному потоку передається одне з можливих значень параметра циклу.

Режим *schedule* забезпечує іншу дисципліну розподілу ітерацій між потоками. Параметр «тип» режиму *schedule* задає один з наступних розподілів ітерацій:

• *static* – блочно-циклічний розподіл ітерацій циклу. Перший блок з *L* ітерацій виконує нульовий потік, другий блок – наступний потік і т. д. до останнього потоку, потім розподіл знову починається з нульового потоку. Якщо розмір не вказаний, то усі ітерації діляться на безперервні частини приблизно однакового розміру (конкретний спосіб залежить від реалізації). Отримані порції ітерацій розподіляються між потоками;

• *dynamic* – динамічний розподіл ітерацій з фіксованим розміром блока (спочатку кожен потік отримує *L* ітерацій, за умовчанням 1). Потік, який закінчує виконання своєї порції ітерацій, отримує першу вільну порцію ітерацій. Вивільнені потоки отримують нові порції ітерацій до тих пір, поки всі порції не будуть вичерпані. Остання порція може містити менше ітерацій, ніж всі інші.

• *guided* – динамічний розподіл ітерацій, при якому розмір порції зменшується з деякого початкового значення до величини *L* (за замовчуванням 1) пропорційно кількості ще не розподілених ітерацій, поділеному на кількість потоків, які виконують цикл. Розмір блоку залежить від реалізації. У ряді випадків такий розподіл дозволяє краще розділити

роботу і збалансувати завантаження потоків. Кількість ітерацій в останній порції може виявитися менше *L*;

• *runtime* – спосіб розподілу ітерацій вибирається під час роботи програми за значенням змінної середовища *OMP_SCHEDULE*. Параметр *L* при цьому не задається.

Режим *ordered* дозволяє в циклі використовувати директиву

#pragma omp ordered

Директива *ordered* визначає блок усередині тіла циклу, який повинен виконуватися в тому порядку, в якому ітерації йдуть у послідовному циклі. Цей режим використовується, наприклад, у тому випадку, коли здійснюється виведення даних на екран або у файл.

2.1.4. Використання директиви OMP for

У попередньому прикладі послідовна ділянка має цикл. При цьому обчислення суми добутків елементів рядка масиву *A* на вектор *X* можна здійснювати паралельно. Нижче наведено текст програми з використанням паралельного виконання ітерацій циклу.

Для виконання ітерацій створюється чотири потоки з динамічним розподілом ітерацій. Кінцеве значення змінної *flag* формується з допомогою операції «або».

```
#pragma omp parallel for reduction (|| :flag) num_threads (4) schedule
(dynamic,4)
        for (int i = 0; i < M; i++)
        {
                float ш= 0;
                for (int j = 0; j < M; j++)
                {
                        z = z + a[ i ][ j ] * x [ j ];
                }
```

```
            if (abs(z – Lx [ i ]) > eps)
            {
                flag = 1;
            }
        }
    // закінчення паралельного виконання ітерацій циклу
        if (0 = = flag)
            cout << "Вектор X не є власним" << endl;
        else
            cout << "Вектор X – власний" << endl;
```

2.2. Порядок виконання роботи

1. Розробити програму з послідовним варіантом виконання відповідно до завдання.
2. Виділити ділянки з можливим паралельним виконанням.
3. Визначити можливість паралельного виконання циклів.
4. Вбудувати в послідовний варіант директиви OpenMP.
5. Вставити в текст програми функцію виміру часу виконання.
6. Здійснити трансляцію програми в послідовному варіанті.
7. Зняти час виконання послідовного виконання програми.
8. Налаштувати транслятор системи програмування на використання режиму *OpenMP*.
9. Здійснити трансляцію програми для роботи в паралельному режимі.
10. Зняти час виконання програми в паралельному режимі.
11. Перевірити роботу програми залежно від кількості потоків у паралельних областях.
12. Побудувати графіки для часу виконання програми в послідовному і паралельному режимах при різних розмірах даних. Результати пояснити.

Таблиця 2.1 – Варіанти завдань

Варіант	Завдання
1	Натуральне число з n цифр є числом Армстронга, якщо сума його цифр, зведених у ступінь *n*, дорівнює самому числу (приклад $1^3 + 5^3 + 3^3 = 153$). Отримати всі числа Армстронга, що складаються з трьох або чотирьох цифр.
2	Знайти в проміжку від 1 до 1000 числа, у яких п’ять дільників.
3	Скільки чисел у діапазоні від 1 до n мають усі різні цифри в десятковому поданні?
4	Два натуральних числа називають дружніми, якщо кожне з них дорівнює сумі всіх дільників іншого, крім самого цього числа. Знайти всі пари дружніх чисел, що лежать у діапазоні від n до m.
5	Перевести числа, що зберігаються в масиві, з десяткової системи числення у восьмеричне подання.
6	Знайти в діапазоні від n до m числа, у яких парне число дільників.
7	Для чисел з діапазону 1 до 10 000 знайти, скільки разів у кожному числі зустрічається кожна цифра.
8	Натуральне число називається досконалим, якщо воно дорівнює сумі всіх своїх дільників за винятком самого себе. Отримати всі досконалі числа, менші *n* і більші *m*.
9	Обчислити добуток двох матриць. Розмірності матриць вводяться з клавіатури, введення матриць здійснюється з файлів: перша матриця з файлу *A.txt*, а друга з *B.txt*
10	Визначити радіус і центр кола, на якій лежить найбільше число точок заданого на площині масиву точок $A(x_i,y_i)$ i=1,100.

Контрольні запитання

1. За допомогою якої функції можна отримати число потоків у паралельній області?

2. Як отримати номер поточного потоку?

3. Яке значення повертає функція *omp_get_num_threads*( ), якщо режим *OpenMP* не встановлений?

4. Як за допомогою номера потоку та загальної кількості запущених потоків здійснити розподіл роботи з виконання програми?

5. Для чого використовується директива *sections*?

6. За допомогою якого параметра можна встановити число потоків, які виконують окремі секції?

7. Що визначає директива *section*?

8. Які номери отримують потоки, що виконують окремі секції?

9. Які обмеження накладаються на цикли, ітерації яких можна виконувати паралельно?

10. Які операції допустимі при завданні умови закінчення паралельного циклу?

11. Чи може змінюватися індексна змінна циклу в тілі циклу при організації паралельного виконання?

12. Чому дорівнює змінна паралельного циклу після закінчення всіх ітерацій?

13. Які існують варіанти розподілу ітерацій паралельного циклу між потоками виконання?

3. СИНХРОНІЗАЦІЯ ПОТОКІВ В OPENMP

Мета роботи: освоєння прийомів синхронізації виконання потоків з використанням директив *OpenMP.*

3.1. Синхронізація потоків в OpenMP

При організації багатопотокового застосування виникає проблема коректного використання загальних ресурсів.

Так, наприклад, при записі значення в глобальну змінну з боку різних потоків результат може бути непередбачуваним.

Крім того, може виникати ситуація, коли продовження роботи програми можливе тільки після закінчення всіх потоків, запущених раніше. Наприклад, якщо потоки формують значення окремих елементів масиву.

Для вирішення таких проблем використовується синхронізація виконання потоків.

3.1.1. Бар'єр

Найпоширеніший спосіб синхронізації в *OpenMP* – бар'єр. Він реалізується з допомогою директиви *barrier*

#pragma omp barrier .

Потоки, що виконують поточну паралельну область, дійшовши до цієї директиви, зупиняються і чекають, поки всі потоки не дійдуть до цієї ж точки програми, після чого потоки розблоковуються і продовжують своє виконання далі.

3.1.2. Критичні секції

За допомогою директиви *critical* оформляється критична секція програми.

#pragma omp critical [(ім'я)]

{ // Блок операторів критичної секції

}

У кожен момент часу у критичній секції може перебувати не більше одного потоку. Якщо критична секція вже виконується яким-небудь потоком, то всі інші потоки, які виконали директиву для секції з даним ім'ям, будуть заблоковані, поки потік, якій увійшов раніше у секцію, не закінчить виконання даної критичної секції. Як тільки працюючий потік вийде з критичної секції, один із заблокованих на вході потоків увійде в секцію. Якщо на вході в критичну секцію є декілька потоків, то випадковим чином вибирається один з них, а інші потоки продовжують режим очікування.

Критична секція може мати ім'я. Критичні секції, що мають теж саме ім'я, розглядаються як єдина секція, навіть якщо знаходяться в різних паралельних областях.

Всі неіменовані критичні секції умовно асоціюються з тим самим іменем.

Вхід у критичну секцію, а також вихід з неї має здійснюватися через перший і відповідно останній оператор блока. Побічні входи і виходи з критичної секції (наприклад, за допомогою оператора goto) заборонені.

Нижче наведений приклад використання критичної секції для синхронізації потоків.

```
#include <stdio.h>
#include <omp.h>
int main ( int argc, char *argv [ ] )
{
  int n;
#pragma omp parallel
  {
#pragma omp critical
   {
     n=omp_get_thread_num ( ) ;
     printf("Потік %d\n", n);
```

```
  }
 }
}
```

Критичні секції додають послідовні ділянки коду в паралельну програму, що може знизити її ефективність.

3.1.3. Директива atomic

Частим випадком використання критичних секцій на практиці є оновлення загальних змінних. Наприклад, якщо змінна *sum* є загальною і оператор виду *sum=sum+expr* знаходиться в паралельній області програми, то при одночасному виконанні цього оператора декількома потоками можна отримати некоректний результат. Щоб уникнути такої ситуації, можна скористатися механізмом критичних секцій або спеціально передбаченою для таких випадків директивою *atomic*.

```
#pragma omp atomic
```

Дана директива належить до оператора присвоювання, що йде безпосередньо за нею, гарантуючи коректну роботу з загальної змінної, що стоїть в його лівій частині. На час виконання оператора блокується доступ до даної змінної всім запущеним у даний момент потокам, крім потоку, що виконує операцію. Атомарною є тільки робота зі змінною в лівій частині оператора присвоювання, при цьому обчислення правої частини не зобов’язані бути атомарними.

```
#include <stdio.h>
#include <omp.ш>
int main ( int argc, char *argv [ ] )
{
  int count = 0;
#pragma omp parallel
   {
```

```
#pragma omp atomic
    count++;
  }
  printf("Кількість потоків: %ш\n", count);
}
```

3.1.4. Блокатори

Один з варіантів синхронізації в *OpenMP* реалізується через механізм блокаторів (*locks*). Як блокатор виступає змінна з загальним доступом з боку всіх потоків (зазвичай така змінна оголошується як глобальна). Для використання блокаторів у програмі оголошується змінна типу *omp_lock_t*. Наприклад:

```
omp_lock_t    locker;
```

Блокатор може перебувати в одному з трьох станів:

• не ініціалізований – стан не визначено;

• розблокований (вільний) – блокування відсутнє, дозволено продовження виконання потоку;

• заблокований (зайнятий) – блокування включене, потік повинен чекати до моменту розблокування.

Блокатор у вільному стані може бути захоплений тільки одним потоком. При цьому блокатор переходить у стан «зайнятий». Звільнити блокатор може тільки потік, який захопив його.

Якщо блокатор не ініціалізований, він не може використовуватися для синхронізації потоків.

3.1.4.1. Типи блокаторів

В *OpenMP* передбачено два типи блокаторів: прості і багаторазові. Багаторазовий блокатор може захоплюватися тим самим потоком перед його звільненням декілька разів, у той час як простий – може бути захоплений тільки один раз.

Для багаторазового блокатора вводиться поняття коефіцієнта захоплення. Спочатку коефіцієнт захоплення встановлюється рівним нулю, при кожному наступному захопленні він збільшується на одиницю, а при кожному звільненні – зменшується на одиницю. Блокатор вважається розблокованим, якщо його коефіцієнт захоплення дорівнює нулю.

3.1.4.2. Ініціалізація блокаторів

Блокатор повинен бути ініціалізований перед першим використанням. Для ініціалізації простого або багаторазового блокатора використовуються відповідні функції:

void omp_init_lock(*omp_lock_t *lock*);

void omp_init_nest_lock(*omp_nest_lock_t *lock*).

Після виконання функції блокатор переводиться в розблокований стан. Для багаторазового блокатора коефіцієнт захоплення установлюється в нуль.

Якщо необхідність у використанні блокаторів відпала, їх слід перевести у неініціалізований стан.

Функції *omp_destroy_lock*() і *omp_destroy_nest_lock*() використовуються для переведення простого або багаторазового блокатора в неініціалізований стан.

void omp_destroy_lock(*omp_lock_t *lock*);

void omp_destroy_nest_lock(*omp_nest_lock_t *lock*);

Якщо блокатор знаходиться в неініціалізованому стані, він не може використовуватися для синхронізації потоків.

3.1.4.3. Захоплення і звільнення блокатора

Для захоплення блокатора використовуються функції

void omp_set_lock(*omp_lock_t *lock*); // простий

void omp_set_nest_lock(*omp_nest_lock_t *lock*); // багаторазовий

Викликавши цю функцію, потік чекає звільнення блокатора, а потім захоплює його. Блокатор переводиться в зайнятий стан.

Якщо багаторазовий блокатор вже захоплений даним потоком, то потік не блокується, а коефіцієнт захоплення збільшується на одиницю.

Для звільнення блокатора використовуються функції:

void omp_unset_lock (*omp_lock_t *lock*);

void omp_unset_nest_lock (*omp_lock_t *lock*);

Виклик функції *omp_unset_lock* звільняє простий блокатор, якщо він був захоплений цим потоком. Для багаторазового блокатора функція *omp_unset_nest_lock* зменшує на одиницю коефіцієнт захоплення. Якщо коефіцієнт дорівнюватиме нулю, блокування знімається.

Якщо після звільнення блокатора є потоки, які очікують цього звільнення, блокатор буде відразу ж захоплений одним з потоків (невідомо, який саме потік забезпечить захоплення). Потік, отримавший блокатор, продовжить своє виконання далі, інші потоки залишаться в заблокованому стані.

```
#include <stdio.h>
#include <omp.h>
omp_lock_t lock;
int main(int argc, char *argv [ ] )
{
  int n;
  omp_init_lock(&lock);
#pragma omp parallel private (n)
  {
   n=omp_get_thread_num ( ) ;
   omp_set_lock(&lock);
   printf("Початок секції, потік %d\n", n);
   sleep(5);
   printf("Закінчення секції, Потік %d\n", n);
   omp_unset_lock(&lock);
```

```
  }
  omp_destroy_lock(&lock);
}
```

Для перевірки стану блокатора можна використовувати функції:

int omp_test_lock (*omp_lock_t *lock*);

int omp_test_nest_lock (*omp_lock_t *lock*);

Якщо блокатор знаходиться в заблокованому стані, повертається нульове значення. При цьому виконання потоку не зупиняється.

Якщо блокатор вільний, то для простого блокатора функція *omp_test_lock* повертає 1, а для багаторазового – новий коефіцієнт захоплення. В обох випадках блокатор переводиться у стан захвата, а потік, отримавший блокатор, продовжує своє виконання.

3.1.5. Директива flush

При виконанні потоку значення деяких змінних може тимчасово перебувати в регістрах центрального процесора або в кеш-пам’яті поточного потоку. Якщо такі змінні є загальними для декількох потоків, потрібна гарантія зберігання поточного значення в оперативній пам’яті обчислювача.

Для цих цілей і призначена директива *flush*.

#pragma omp flush [(список)]

Виконання даної директиви передбачає, що значення всіх змінних (або змінних зі списку, якщо він заданий), які тимчасово зберігаються в регістрах і кеш-пам’яті поточного потоку, будуть занесені в основну пам’ять; всі зміни змінних, зроблені потоком під час роботи, стануть видимі іншим потокам; якщо якась інформація зберігається у буферах висновку, буфери будуть скинуті у файли і т. п.

При цьому операція проводиться тільки з даними поточного потоку, дані, змінені іншими потоками, не зачіпаються.

До повного завершення операції збереження даних жодні дії з перерахованими в ній змінними здійснюватися не можуть.

Неявно *flush* без параметрів присутня в директиві *barrier*, на вході і виході областей дії директив *parallel, critical, ordered*, на виході областей розподілу робіт, якщо не використовується опція nowait, у викликах функцій *omp_set_lock*(), *omp_unset_lock*(), *omp_test_lock*(), *omp_set_nest_lock*(), *omp_unset_nest_lock*(), *omp_test_nest_lock*(), якщо при цьому блокатор установлюється або знімається. Крім того, *flush* викликається для змінної, яка бере участь в операції, асоційованої з директивою *atomic*.

Директива *flush* не застосовується на вході області розподілу робіт, а також на вході і виході області дії директиви *master*.

3.2. Додаткові функції *OpenMP*

3.2.1. Функція omp_get_level()

Функція *int omp_get_level* ( ) видає кількість вкладених паралельних областей в даному місці коду.

У послідовній області функція повертає значення 0.

3.2.2. Функція omp_get_ancestor_thread_num()

Функція *omp_get_ancestor_thread_num*(*level*) повертає для рівня вкладеності паралельних областей, заданого параметром *level*, номер потоку, який породив цей потік.

int omp_get_ancestor_thread_num(int level);

Якщо *level* менше нуля або більше поточного рівня вкладеності, функція поверне –1. Якщо *level*=0, функція поверне 0. Виклик функції без аргументів *level=omp_get_level*() еквівалентний виклику функції *omp_get_thread_num* ( ).

3.2.3. Функція omp_get_team_size()

Функція *int omp_get_team_size*() повертає для заданого параметром *level* рівня вкладеності паралельних областей кількість потоків, породжених одним батьківським потоком.

Якщо *level* менше нуля або більше поточного рівня вкладеності, повертається -1. Якщо *level*=0, функція поверне 1, а якщо параметр *level* відсутній, виклик еквівалентний виклику функції *omp_get_num_threads*().

3.2.4. Функція omp_get_active_level()

Функція *omp_get_active_level*() повертає для викликів потоку кількість вкладених паралельних областей, оброблюваних більш ніж одним потоком, у даному місці коду.

int omp_get_active_level(void);

При виклику з послідовної області функція повертає значення, рівне нулю.

3.3. Приклад розробки програми з використанням синхронізації між потоками

Необхідно розробити програму для здійснення замовлень для відпочинку на курорті.

Є список будинків із зазначенням вартості у форматі:

ім’я курорту, рівень обслуговування, вартість на одну особу без харчування і з харчуванням, вартість на двох без харчування і з харчуванням.

Наприклад:

Курортна 5 1000 1500 1500 2500

Данні зберігаються у файле *fin.txt*.

Закази зберігаються у файле *zakaz.txt* у форматі:

прізвище замовника, дата замовлення і тривалість, найменування курорту, кількість місць, харчування.

Наприклад:

Іванов 2/04/2016 3 доби курортна одномісний харчування

Нижче наведено текст програми для обробки даних про замовлення.

Для роботи з курортами є структура *kurort*. До складу структури входить блокатор для обмеження доступу з боку декількох потоків і відомості про будинок.

Відомості про замовлення зчитуються в масив структур *Zakazchik*.

Оскільки одного замовника можна обробляти незалежно від іншого, можна організувати паралельне виконання циклу читання відомостей про заказ і пошук курорту.

Директива *pragma omp parallel for num_threads* (*Nz*) private (*s1, dn, s2, Mn, Gn*) забезпечує створення *Nz* потоків для виконання ітерацій циклу обробки даних. Для кожного потоку створюються локальні копії змінних *s1, dn, s2, Mn, Gn*. При резервуванні будинка для замовника використовується блокатор *Spisok*[i].lock, де *i* – номер курорту.

Для роботи з файлами використовується критична секція, яка забезпечує коректне зчитування і запис даних.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <omp.h>
using namespace std;

struct kurort{
    omp_lock_t lock;
    char name[48];
    int zwezda;
    int one;
    int one_food;
    int two;
```

```
        int two_food;
        int free[366];
};
struct Zakazchik
{
        char name[48];
        char data[32];
        int kol_d;
        char d[12];
        char Kur[48];
        char mesto[48];
        char pitanie[48];
        char *zakaz;
        int Summa;
};
int Nz = 4;
int dni[] = {0,31,60,91,121,152,182
                ,213,244,274,305,335};// номер дня для кожного місяця
int main()
{
        ifstream fin;
        ofstream fout;
        kurort *Spisok;
        Zakazchik *Person=new Zakazchik[Nz];
        fout.open("fout.txt");
        fin.open("fin.txt");
        if (!fin.is_open())
        {
                cerr << "File kurort no open" << endl;
```

```
            _wsystem(L"Pause");
            exit(1);
        }
        int n;
        fin >> n;
        Spisok = new kurort[n];
        for (int i = 0; i < n; i++)
        {
            fin >> Spisok[i].name >> Spisok[i].zwezda;
                >> Spisok[i].one>>Spisok[i].one_food
                >>Spisok[i].two>>Spisok[i].two_food;
            for (int j = 0; ш < 366; j++)
                Spisok[i].free[j] = 0;
            omp_init_lock(&(Spisok[i].lock));
        }
        fin.close();
        fin.open("zakaz.txt");
        char *zkz1 = " забронювати ";
        char *zkz2 = "Отказ";
        char *s1,*s2;
        int dn, int Mn, Gn;
        while (!fin.eof())
        {
#pragma omp parallel for num_threads (Nz) private (s1, dn, s2, Mn, Gn)
            for (int j = 0; j < Nz; j++)
            {
#pragma omp critical (FILE)
                {
                    fin >> Person[j].name >> Person[j].data
```

```
                >> Person[j].kol_d >> Person[j].d
                 >> Person[j].Kur >> Person[j].mesto
                >> Person[j].pitanie;
}
if (strlen(Person[j].pitanie)==0)  continue;
s1 = strchr(Person[j].data, '/');
*s1 = 0;
s1++;
dn = atoi(Person[j].data);
s2 = strchr(s1, '/');
*s2 = 0;
s2++;
Mn = atoi(s1);              *( - -s1) = '/';
Gn = atoi(s2);              *( - -s2) = '/';
int L;
L = dni[Mn] + dn;
for (int i = 0; i < n; i++)
{
        if (strcmp(Spisok[i].name, Person[j].Kur) == 0)
        {
        Person[j].zakaz = zkz1;
        if (!strcmp(Person[j].mesto, " одномісна "))
                if (!strcmp(Person[j].pitanie, " харчування "))
        Person[j].Summa = Person[j].kol_d*Spisok[i].one_food;
                else
        Person[j].Summa = Person[j].kol_d*Spisok[i].one;
        else
                if (!strcmp(Person[j].pitanie, "харчування"))
        Person[j].Summa = Person[j].kol_d*Spisok[i].two_food;
```

```
                    else
                Person[j].Summa = Person[j].kol_d*Spisok[i].two;
                    omp_set_lock(&(Spisok[i].lock));
                    for (int k = 0; k < Person[j].kol_d; k++)
                    {
                        if (Spisok[i].free[L + k] == 0)
                            Spisok[i].free[L + k] = 1;
                        else
                        {
                            Person[j].zakaz = zkz2;
                            Person[j].Summa = 0;
                            while (k > 0)
                            {
                                Spisok[i].free[L + k] = 0;
                                k--;
                            }
                            break;
                        }
                    }
                    omp_unset_lock(&(Spisok[i].lock));
                    break;
                }
            }
#pragma omp critical (FILE1)
            {
            fout << Person[j].name << ' ' << Person[j].data << ' '
                << Person[j].kol_d << ' ' << Person[j].d
                << ' ' << Person[j].Kur << ' ' << Person[j].mesto
                << ' ' << Person[j].pitanie << ' ' << Person[j].Summa
```

```
                    << ' ' << Person[j].zakaz << endl;
                }
            }
        }
        fin.close();
        fout.close();
    return 0;
}
```

3.4. Порядок виконання роботи

1. Розробити програму з послідовним варіантом виконання у відповідності з завданням.
2. Виділити ділянки з можливим паралельним виконанням.
3. Визначити можливість паралельного виконання циклів.
4. Забезпечити синхронізацію при роботі з файлами і доступ до загальних змінних.
5. Вбудувати в послідовний варіант директиви *OpenMP*.
6. Вставити в текст програми функцію виміру часу виконання.
7. Здійснити трансляцію програми в послідовному варіанті.
8. Зняти час виконання послідовного виконання програми.
9. Налаштувати транслятор системи програмування на використання режиму *OpenMP*.
10. Здійснити трансляцію програми для роботи в паралельному режимі.
11. Зняти час виконання програми в паралельному режимі.

Варіанти завдань

1. У файлі задано плоскі фігури у формі багатокутників з координатами вершин. Для кожної фігури підрахувати площу і обчислити загальну площу всіх фігур.

2. У файлі задані коефіцієнти декількох поліномів. Знайти корені кожного полінома і підрахувати загальну кількість коренів, які більше нуля.

3. У файлі дано дійсні додатні числа a, b, c, d. З'ясувати, чи можна прямокутник зі сторонами a, b помістити в прямокутник зі сторонами c, d так, щоб сторони одного прямокутника були паралельні або перпендикулярні сторонам іншого прямокутника.

4. У файлі дані відомості про телефонні розмови: код міста, звідки дзвонять, код міста, куди здійснювався дзвінок, тривалість дзвінка, номер телефону. Написати програму, яка за кодом міста і тривалості переговорів обчислює вартість переговорів і для кожного абонента формує загальну вартість. Результат вивести у файл.

5. У файлі подані прізвища працівників та дати народження кожного працівника. Дата народження задана у вигляді: рік, назва місяця і номер дня місяця. Написати програму, яка за датою народження (день і місяць) визначає знак Зодіаку. Підрахувати загальну кількість працівників, народжених під кожним знаком.

6. У файлі дано відомості про замовлення на авіарейси у форматі: місто, аеропорт, час, день, місяць. Якщо квиток замовляється за 45 діб – знижка 20 %, за 20 діб – знижка 10 %. Визначити вартість замовлення. Підрахувати загальну вартість замовлень для кожного аеропорту призначення.

7. У файлі задано список студентів і оцінки з предметів у формі: прізвище, номер студентського квитка, назва предмета, оцінка. Предметів і оцінок з предмета може бути декілька. Для кожного студента підрахувати рейтинг (середній бал). Якщо студент з будь-якого предмета має незадовільну оцінку або не має оцінки – рейтинг не підраховується.

8. У файлі задано список страв. Для кожної страви є назва, список інгредієнтів. Кожен інгредієнт заданий у вигляді: назва та кількість (у грамах). Підрахувати калорійність страв та необхідну кількість кожного інгредієнта для всіх страв.

9. У файлі зберігаються відомості про надсилання повідомлень з поштового сервера: e-*mail* відправника; e-*mail* одержувача; дата відправлення; час відправлення; розмір повідомлення (Кб). Для кожного відправника отримати загальний розмір повідомлень і вартість.

10. У файлі зберігаються відомості про виконані роботи: назва роботи; категорія роботи; виділена кількість годин на виконання даної роботи; реально витрачений час на виконання роботи; базова вартість роботи. Якщо витрачений час менше запланованого більш ніж на два дні, виписується премія. Якщо витрачений час більше запланованого більш ніж на два дні, виписується штраф. Розмір штрафу і премія залежать від категорії роботи. Для кожної роботи підрахувати остаточну вартість, загальну премію і штраф за весь комплекс робіт.

Контрольні запитання

1. Чому може знадобитися синхронізація виконання потоків?
2. В якому випадку використовується синхронізація за допомогою директиви *barrier*?
3. Що таке критична секція і як вона оформлюється у тексті програми?
4. Чи може критична секція мати ім’я?
5. Як розглядаються критичні секції, що мають одне ім’я?
6. Чи можна передавати управління у критичну секцію ззовні на довільний оператор (оператор може мати мітку)?
7. Чи можна закінчувати критичну секцію не через останній оператор?

8. Для чого використовується директива *atomic*?

9. На яку частину оператора присвоювання поширюється дія директиви atomic?

10. Що собою являє блокатор?

11. В яких станах може перебувати блокатор?

12. Які типи блокаторів підтримує *OpenMP*?

13. За допомогою якої функції встановлюється початкове значення блокатора?

14. Як здійснити перевірку стану блокатора?

15. Для чого використовується директива *flush*?

СПИСОК ЛИТЕРАТУРИ

1. Воеводин В.В. Параллельные вычисления./ В.В. Воеводин, Вл.В Воеводин. СПб.: БХВ-Петербург, 2002. – 608 с.

2. Barbara Chapman. Using OpenMP: portable shared memory parallel programming (Scientific and Engineering Computation)./ Barbara Chapman, Gabriele Jost, Ruud van der Pas. Cambridge, Massachusetts: The MIT Press., 2008. - 353 pp.

3. Антонов А.С. Введение в параллельные вычисления. Методическое пособие. / А.С. Антонов. – М.: Изд-во Физического факультета МГУ, 2002. – 70 с.

4. OpenMP Architecture Review Board [Електроний ресурс] Режим доступу: // http://www.openmp.org/.

5. The Community of OpenMP Users, Researchers, Tool Developers and Providers [Електроний ресурс] Режим доступу:// http://www.compunity.org/.

6. OpenMP Application Program Interface Version 3.0 May 2008 [Електроний ресурс] Режим доступу:// http://www.openmp.org/mp-documents/spec30.pdf.

7. Что такое OpenMP? [Електроний ресурс] Режим доступу:// http://parallel.ru/tech/tech_dev/openmp.html.

ЗМІСТ

Навчальне видання

Методичні вказівки
до лабораторних робіт
“ Розробка програм для симетричних багатопроцесорних обчислювальних систем з використанням OPENMP у середовищі VISUAL STUDIO ”
з дисципліни “ Архітектура обчислювальних систем ”
для студентів спеціальностей
122 – комп’ютерні науки та інформаційні технології,
124 – системний аналіз,186 – видавництво та поліграфія.

Укладачі:
КОЖИН Юрій Миколайович
МАЛИХ Олег Миколайович
ПРОКОПЕНКОВ Володимир Пилипович

Відповідальний за випуск *О.С. Куценко*
Роботу до друку рекомендував *М.І. Безменов*

План 2017 , поз. 134

Підп. до друку.	Формат 60х84 1/16.	Папір офсетний.
Riso–друк.	Гарнітура Таймс.	Ум. друк. арк. 2,45.
Наклад 25 прим.	Зам. №	Ціна договірна.

Видавець
Видавничий центр НТУ “ХПІ” м. Харків, 61002, вул. Кирпичова, 2
Свідоцтво про державну реєстрацію ДК №4064 від 21.08.2017 р